Editor: Martin Fowler ■ ThoughtWorks ■ fowler@acm.org

Public versus Published Interfaces

Martin Fowler

One of the growing trends in software design is separating interface from implementation. The principle is about separating modules into public and private parts so that you can change the private part without coordinating with other modules. However, there is a further distinction—the one between public and published interfaces. This distinction is important because it affects how you work with the interface.

Public versus published

Let's assume I'm writing an application in a modern modular language—to make things more concrete, let's assume this language is Java. My application thus consists of several classes (and interfaces), each of which has a public interface. This public interface of a class defines a group of methods that any other class in the system can invoke.

While I'm enhancing a public method, I realize that one of its parameters is redundant—I don't need to use it in the method (maybe I can get that value through some other route or maybe I just don't need it anymore). At this point, I can eliminate that value from the method signature, clarifying the method and potentially saving work for its callers. Because this method is public, any class in the system can call it. Should I remove the parameter?

In this case, I would argue yes, because there are benefits and it isn't difficult. Although the method might be used anywhere, I can easily find the users with a search tool. If I have one of the new breeds of refactoring tools (see www.refactoring.com for details) available for Java, I can do it with a simple menu click—the tool will then automatically update all the callers. So, changing a public method isn't a big deal.

However, things rapidly change if I put that software out on the Web as a component, and other people, whom I don't know, start building applications on top of it. If I now delete the parameter, everybody else's code will break when I upgrade. Now I must do something a little more elaborate. I can produce the new method with one less parameter but keep the old method—probably recoding the old method to call the new one. I mark the old method as deprecated, assuming people will move the code over and that I can change it in the next release or two.

The two cases are quite different, yet there's nothing in the Java language to tell the difference—a gap that's also present in a few other languages. Yet there's something to be said for the public–published distinction being more important than the more common public–private distinction.

The key difference is being able to find and change the code that uses an interface. For a published interface, this isn't possible, so you need a more elaborate interface update process. Interface users are either callers or are classes that subclass or implement an interface.

0740-7459/02/$17.00 © 2002 IEEE

Advice on publishing

Recognizing the difference between public and published leads to an important set of consequences.

Don't treat interfaces as published unless they are

If you need to change an interface and can find and change all users, then don't bother going through all the forwarding and deprecation gambits. Just make the change and update the users.

Don't publish interfaces inside a team

I once suggested to somebody that we change a public method, and he objected because of the problems caused by its being published. The real problem was that although there were only three people on the team, each developer treated his interfaces as published to the other two. This is because the team used a strong form of code ownership in which each module was assigned to a single programmer and only that programmer could change the module's code. I'm sympathetic to code ownership—it encourages people to monitor their code's quality—but a strong code ownership model such as this one causes problems by forcing you to treat interperson interfaces as published.

I encourage a weaker ownership model in which one person is responsible for the module but other people can make changes when necessary. This lets other developers do things such as alter calls to changed methods. (You can also use collective code ownership—where anyone can change anything—to avoid internal publishing.) This kind of ownership usually requires a configuration management system that supports concurrent writers (such as CVS) rather than one that uses pessimistic locking.

There is a limit to how big a team you can run without some form of internal publishing, but I would err on the side of too little publishing. In other words, assume you don't need to publish interfaces, and then adjust if you find this causes problems.

Publish as little as you can as late as you can

Because publishing locks you into the slower cycle of changes, limit how much you publish. This is where a language's inability to distinguish between public and published becomes an issue. The best you can do is declare some modules to be the interface and then counsel your software users not to use the other modules, even if they can see them. Keep these interfaces as thin as you can. Publish as late as possible in the development cycle to give yourself time to refine the interfaces. One strategy is to work closely with one or two users of your components—users who are friendly enough to cope with sharp interface changes—before you publish to the masses.

Try to make your changes additions

In addition to distinguishing between published and public interfaces, we can also identify two types of interface changes. Generally, changes can alter any aspect of an interface. However, there are some changes that only cause additions to an interface, such as adding a method. Additions won't break any of the interface's clients—existing clients have no problem using the old methods. Consequently, when you make a change, it's worth considering whether you can recast it into an addition. For example, if you need to remove a parameter from a method, instead of changing the method, try adding a new method without the parameter. That way, you get an addition rather than a general alteration, and your clients remain compatible.

Additions can still cause problems if outside groups have their own implementation of your interface. If that happens, even adding a method breaks the alternative implementation. Thus, some component technologies, such as COM, use immutable interfaces. With an immutable interface, once it's published, you guarantee not to change it. If you want to change the interface, you must create a second interface, and components can then support this interface at their leisure. It's not the ideal scheme, but it certainly has its merits.

There's something to be said for the public–published distinction being more important than the more common public–private distinction.

I would like to see the public–published distinction appear more in languages and platforms. It's also interesting that environments don't tend to provide the facilities to evolve interfaces. Some can deprecate a method that's due to be removed: Eiffel does this as part of the language, and Java does it (but as part of the built-in documentation). I haven't seen anyone add a marker to a method that warns implementers of something that's going to be added or would let you add something to an interface in a provisional way.

That's part of a more general issue in software platforms. So far, platforms haven't sufficiently understood that software is supposed to be soft and thus needs facilities that allow change. In recent years, we've taken more steps in this direction with component-packaging systems, but these are just the early steps.

Martin Fowler is the chief scientist for ThoughtWorks, an Internet systems delivery and consulting company. Contact him at fowler@acm.org.